Introduction to C Language

Instructor: Professor I. Charles Ume
ME 6405

Introduction to Mechatronics

Fall 2006

Instructor: Professor Charles Ume

Introduction to C Language
History of C Language

- In 1972, the Unix Operating system was being developed. During this time, the concept of a system programming language having attributes of both “low level” and “high level” languages also developed. System programming languages have advantages of a high level language but allow the programmer to take control of the underlying hardware if desired.

- Brian W. Kernighan and Dennis M. Ritchie developed C at Bell Laboratories as a system programming language. Their underlying goal was to develop a language that was simple and flexible enough to be used in a variety of different processors.

- In 1983, the American National Standards Institute(ANSI) established a committee with a goal to produce “ an unambiguous and machine-independent definition of the language C”. This ANSI C was completed in 1988.

- Currently, C and C++ are the most popular computer languages. (Note: C++ is an object oriented version of C)
References

Texts:

Websites:
http://cm.bell-labs.com/cm/cs/who/dmr/ctut.pdf
(Note: this is a tutorial on C written by Kernighan himself)
C in relationship to other languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Level</th>
<th>Ease of Programming</th>
<th>Relative Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>High</td>
<td>Easy</td>
<td>Slow</td>
</tr>
<tr>
<td>Pascal</td>
<td>High</td>
<td>Fairly easy</td>
<td>Moderate</td>
</tr>
<tr>
<td>Fortran</td>
<td>High</td>
<td>Moderate</td>
<td>Moderate</td>
</tr>
<tr>
<td>C</td>
<td>Mid</td>
<td>Fairly difficult</td>
<td>Fast</td>
</tr>
<tr>
<td>Assembly</td>
<td>Low</td>
<td>Difficult</td>
<td>Very fast</td>
</tr>
</tbody>
</table>
Operators  
(Note: C is case sensitive)

Assignment Operators

= Assignment x = y;

Mathematical Operators

+ Addition x = x + y;
- Subtraction x = x – y;
* Multiplication x = x*y;
/ Division x = x/y;
% Modulus x = x %y
++ Increment x++; 
-- Decrement x--;

(Note: Modulus is the same as remainder from division
Example: x=7%3 => x=1)
Operators

Relational and Logical Operators

== Equal to
!= Not Equal to
< Less than
> Greater than
<= Less than or equal to
>= Greater than or equal to

&& Logical AND
|| Logical OR

(Note: When using the logical operations, any nonzero value is considered “true.” Only 0 value is considered false)
Operators

Bitwise Operators

& Bitwise AND
| Bitwise inclusive OR
^ Bitwise exclusive OR
~ Bitwise NOT
>> Shift Bits Right
<< Shift Bits Left

<table>
<thead>
<tr>
<th>a</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

| and | a & b | 0 | 0 | 0 | 1 |
| or | a | b | 0 | 1 | 1 | 1 |
| exclusive or | a ^ b | 0 | 1 | 1 | 0 |
| one's complement | ~a | 1 | 1 | 0 | 0 |

Examples:

y = 0x0A & 0x02 => y = 0x02
z = 26 ^ 9 => z = 19
r = 23 >> 2 => r = 5

(Note: if 0x prefix is used before a constant, the constant is hexadecimal)

(Note: One must distinguish the bitwise operations & and | from the logical operators && and ||. For example, if x is 1 and y is 2, then x & y is zero while x && y is one.)
Separators

{-} These separators are used to group together lines of code and initialize arrays.

; This separator is used to separate lines of code.

(  ) These separators are used to specify parameters and to change mathematical operator precedence.

[  ] These separators are used to index arrays.

“ ” These separators indicate a string constant.

‘ ’ These separators indicate a character constant.

/* */ These separators indicate a comment. You can also use // to add a single line comment.

Example: Changing mathematical operator precedence:

\[ 5 + 7 \times 2 = 19 \]

\[ ( 5 + 7 ) \times 2 = 24 \]
### Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size in Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>0</td>
<td>nothing</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>a single byte (signed or unsigned)</td>
</tr>
<tr>
<td>int</td>
<td>2</td>
<td>one word (signed or unsigned)</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>long integer, at least 32 bits (signed or unsigned)</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>single-precision floating point</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>double-precision floating point</td>
</tr>
</tbody>
</table>

(Note: The size in bytes a data type uses varies between every development system so be careful. They are machine/compiler dependent)
Declaring a variable

- Before any variable is used in a C program, it must be declared.
  Example: Declaring a integer variable x
  ```
  int x;
  ```

- An initial value can be assigned during a variable declaration.
  Example: Declaring a float variable y with initial value 6.1
  ```
  float y = 6.1;
  ```

- A variable declared using a lower precision can be assigned to a variable declared using a higher precision
  Example: Assigning x to y
  ```
  x = 4;
  y = x; /* y now contains 4.0 */
  ```

- A variable declared using a higher precision can be assigned to a variable declared using a lower precision by “type casting.” “Type casting” temporarily re-declares the type of variable.
  Example: Assigning y to x
  ```
  x = (int)y; /* y is temporarily re-declared as int and assigned to x. The re-declaration of a float to int has the effect keeping only the integer part. Therefore, x now contains 6*/
  ```
An *array* is a collection of values. An array is allocated using the [] separator.

**Examples:**

1. Declaring an array of integers that holds five values.
   ```java
   int myarray[5];
   ```
2. Filling an array with an initial value.
   ```java
   int myarray[5] = { -300, -400, -2, 4, 5};
   ```
3. Alternative way to declare an array with initial values.
   ```java
   int myarray[] = { -300, -400, -2, 4, 5};
   ```
   (Note: the compiler is smart enough to figure out that it needs an array of 5 to hold all the initial values.)
4. You can declare multi-dimensional arrays
   ```java
   int myarray[3][2] = {{1,2},{3,4},{3,1}};
   ```
Indexing Arrays

To use a value stored in an array, the array must be indexed using the [] separator.

Example: indexing a one-dimensional array

```c
int x;
int myarray[5] = { -300, -400, -2, 4, 5};
x = myarray[1]; /* x now contains -400 */
```

Example: indexing a two-dimensional array

```c
int x;
in myarray[3][2] = {{1,2},{3,4},{3,1}};
x = myarray[1][1]; /* x now contains 4 */
```

(Note: An array starts from index ‘0’ in C language. Therefore in example 1, there was really 6 slots. The reason for this is the last slot of any array contains 0 to terminate the array)
Control Flow

- The first basic method to control the flow of a program is by using `if/else`

Syntax:     if  (condition) {
              \_execute this code if condition is true
            }
            else {
              \_execute this code if condition is false (Note: if there is no
code here, else can be omitted)
            }

Example:

If  (a > b) {
    z = a;
}
else {
    z = b;
}
Control Flow

- if/else statements can be chained together

Syntax:    
```java
    if (condition1){
        \text{execute this code if condition 1 is true}
    }
    \text{else if (condition2){}
        \text{execute this code if condition 1 is false and condition 2 is true}
    }
    \text{else if (condition3){
        \text{execute this code if condition 1\&2 is false and condition 3 is true}
    }
    \text{else {}
        \text{execute this code if conditions 1,2,\&3 are false}
    }
```

The George W. Woodruff School of Mechanical Engineering
Control Flow

- The second basic method to control the flow of a program is by using `for` or `while` loops

Syntax: `for ( initial; condition; modify) {`
   
   (Note:
   initial: initializes the loop control variable
   condition: loop while the condition on loop control variable is true
   adjust: code to adjust the loop control variable. Executed once per loop)

   execute this code while condition is true

   }`

Example:
```
for ( i = 0; i < 5; i++)
{
   /* code here is executed 5 times */
}
```
It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The _break_ statement causes the innermost enclosing loop to be exited immediately.

Example: /* find the first ‘a’ in a character string*/

```c
int i = 0;
while(mystring[i] != 0){
    if (mystring [i] == ‘a’ ){
        break;
    }
    i++;
}

/* i now contains index of character ‘a’ in mystring */
```
Functions

- Functions are the building blocks of a C program. Functions are sections of code that perform a single, well-defined service.
- Like a variable, functions must be declared before they are used.
- Functions cannot be declared or initialized within another function.

Example: A function declaration

```c
int Multiply(int x, int y);
```

Example: Initializing the declared function

```c
int Multiply(int x, int y) {
    int z;
    z = x * y;
    return z;
}
```

The George W. Woodruff School of Mechanical Engineering
Example: Using the multiply function

```c
int result;
result = Multiply(6,7);  /* result now contains 42 */
```

Example: Use an array as a parameter,

```c
void ArrayFunction(int SomeArray[])
{
  ..
  ..
  return;
}
```

Example: Using the array function

```c
int MyArray[10];
ArrayFunction(MyArray);
```
<table>
<thead>
<tr>
<th>File Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*.c</td>
<td>Source File. They contain functions. If the source file contains a function called “main”, variable declarations outside of any functions (global variables), and function declarations can also be included.</td>
</tr>
<tr>
<td>*.lib</td>
<td>Library file. If the source file will never change then the programmer can create a *library file. The programmer can then use the functions in the library file. A function called “main” is not allowed in a library file.</td>
</tr>
<tr>
<td>*.h</td>
<td>Header File. Every *.c or *.lib file without a function called “main” has a header file. They contain global variable declarations and function declarations for global variables and functions in the *.c or *.lib file.</td>
</tr>
<tr>
<td>*.o</td>
<td>Object File. After a compilation of a source file, an object file is created. This is done primarily to save the compiler time. The compiler will only compile source files that have recently been changed.</td>
</tr>
<tr>
<td>*.exe</td>
<td>Executable program. This is the only file extension that varies from system to system. When a user compiles a program, a compiler creates object files from the source files then a *linker links any functions used in the object files together to create an *executable file. The execution always starts from the function called “main.”</td>
</tr>
</tbody>
</table>
Pre-Processor

Pre-Processor directives are instructions for the compiler. Pre-processor directives are prefixed by the ‘#’ character. Only two different pre-processor commands are discussed here.

#include <filename.h>
The include directive is used to link C source files and library files together. For this discussion we will use this to link the library files. This directive must appear before global variable declarations and function declarations.

#define NAME VALUE
The define directive is used to set definitions. Similar to EQU in assembly.

Example:
#define PI 3.14
Library Functions

Input and Output: #include <stdio.h>

A. Formatted Output

    int printf(const char *format, argument, argument2, etc ...);

    • Accepts a series of arguments
    • Applies to each argument a \textit{format specifier} contained in the \textit{format string} *format
    • Outputs the formatted data to the screen.

Example with no arguments

    printf(“Hello”)
Using arguments and *format specifiers*, we can print a variable’s value to the screen.

**Format Specifiers**

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>Integer signed decimal integer</td>
</tr>
<tr>
<td>%i</td>
<td>Integer signed decimal integer</td>
</tr>
<tr>
<td>%o</td>
<td>Integer unsigned octal integer</td>
</tr>
<tr>
<td>%u</td>
<td>Integer unsigned decimal integer</td>
</tr>
<tr>
<td>%x</td>
<td>Integer unsigned hexadecimal in t (with a, b, c, d, e, f)</td>
</tr>
<tr>
<td>%X</td>
<td>Integer unsigned hexadecimal int (with A, B, C, D, E, F)</td>
</tr>
<tr>
<td>%f</td>
<td>Floating point signed value</td>
</tr>
<tr>
<td>%c</td>
<td>Character Single character</td>
</tr>
<tr>
<td>%s</td>
<td>String pointer Prints characters until a null-terminator</td>
</tr>
<tr>
<td>%%</td>
<td>None Prints the % character</td>
</tr>
</tbody>
</table>

**Example:** Printing a variable’s value

```c
int x = 12;
int y = 10;
int z = x + y;
printf("%d plus %d equals %d",x,y,z)      12 plus 10 equals 22
printf("%X plus %X equals %X",x,y,z) C plus A equals 16
```
To display non-graphical characters (such as a new-line character) and separators, an *escape sequence* is needed.

**Escape Sequences**

\n \n LF LF Newline (linefeed)
\r CR \r Carriage return
\t HT \t Tab (horizontal)
\\ \ \ \\ Backslash
\' \' \' Single quote (apostrophe)
\" \" \" Double quote
\? \? \? Question mark
B. Formatted Input

\[
\text{int scanf(const char *format[, address, ...]);}
\]

- scans a series of input fields one character at a time
- formats each field according to a corresponding format specified in the format string.

**Example: Entering a number**

\[
\begin{align*}
\text{int } & \text{x;} \\
\text{printf(“Please input an integer:”);} \\
\text{scanf(“%d”,&x);} \\
\end{align*}
\]

**Example: Entering a string**

\[
\begin{align*}
\text{char buffer[80];} \\
\text{printf(“Please input a string”);} \\
\text{scanf(“%s”,buffer);} \\
\end{align*}
\]
Mathematical Functions: <math.h>

The header <math.h> declares mathematical functions and macros

\[
\begin{align*}
sin (x) & \quad \text{sine of } x \\
cos (x) & \quad \text{cosine of } x \\
tan (x) & \quad \text{tangent of } x \\
asin (x) & \quad \sin^{-1} (x) \\
acos (x) & \quad \cos^{-1} (x) \\
atan (x) & \quad \tan^{-1} (x) \\
extp (x) & \quad \text{exponential function } e^x \\
log (x) & \quad \text{natural logarithm } \ln(x) \\
pow (x, y) & \quad x^y \\
ceil (x) & \quad \text{smallest integer not less than } x \\
floor (x) & \quad \text{largest integer not greater than } x \\
fabs (x) & \quad \text{absolute value } |x| \\
fmod (x, y) & \quad \text{floating-point remainder of } x/y, \text{ with the same sign as } x.
\end{align*}
\]
Other Functions:

- **ctype.h**  Character Class Tests. It declares functions for testing characters, such as to test whether a letter is in lower-case or upper-case.
- **string.h**  String Functions. Such as copy string, comparing string, etc.
- **stdlib.h**  Utility Functions. It declares functions for number to string conversions, memory storage allocation and similar tasks.
Variable Scope

Any variable declared in a level is only available to higher numbered levels. If a variable is declared in Level 1a, only Level 2a, and Level 2b will have access to the variable. If we declare a variable in the Base Level then it becomes a global variable.

Base Level

{ 
Level 1 a: inside a function
  { 
    Level 2a
  }
  { 
    Level 2b
  }
} 

{ 
Level 1 b: inside a function
}
Return to Variables and Functions

Formally, C is pass by instance, which means that a function receives copies of the values of its parameters.

```c
int multbytwo(int z) {
    z = z * 2;
    return z;
}

/* Somewhere in “main” function */
int z = 5
j = multbytwo(z); /* j contains 10 and z still contains 5*/
```

Exception:
When the argument passed to a function is an array, the function does not receive a copy of the array, and it therefore can modify the array. (Call by reference)
Example: Integer to binary conversion

```c
#include <stdio.h> /* pre-processor commands come first*/
#define true 1

int IntToBinary(int x, int bin[]); /* global variables and function declarations come next*/

void main(void)
{
    int x; /* Declare an integer variable */
    int limit; /* Declare an integer variable */
    int binaryarray[20]; /* Declare an integer array */

    /* Let the user input a variable and make sure it is positive*/
    while(true) {
        printf("Input a positive integer: ");
        scanf("%d", &x);
        if(x < 0) {
            printf("Not a positive integer.");
        } else {
            break;
        }
    }
}
```
/* Call the Int to Binary function to store the bits in binaryarray */
limit = IntToBinary(x,binaryarray);

/* Now print the contents of the binary array. */
for(int counter = 0; counter < limit; counter = counter ++)
    printf(“
 Bit %d is %d”,counter,binaryarray[counter]);

/* End the program*/
return;
}

/* Now define the function IntToBinary */
int IntToBinary(int x, int bin[])  {
    int counter = 0;
    while( x > 0 )  {
        int (x & 0x01)
            bin[counter] = 1;
        else
            bin[counter] = 0;
        x = x >> 1;counter = counter + 1;
    }
    return counter;
}